Distributing Security-Mediated PKI*

Gabriel Vanrenen and Sean Smith

Department of Computer Science/PKI Lab
Dartmouth College, Hanover NH 03755 USA
gabriel.vanrenen@alum.dartmouth.org
sws@cs.dartmouth.edu

Abstract. The SEM approach to PKI (by Boneh et al [4]) offers many advantages, such as instant revocation and compatibility with standard RSA tools. However, it has some disadvantages with regard to trust and scalability: each user depends on a mediator that may go down or become compromised.
In this paper, we present a design that addresses this problem. We use secure coprocessors linked with peer-to-peer networks, to create a network of trustworthy mediators, to improve availability. We use threshold cryptography to build a back-up and migration technique, to provide recovery from a mediator crashing while also avoiding having all mediators share all secrets. We then use strong forward secrecy with this migration, to mitigate the damage should a crashed mediator actually be compromised. We also discuss a prototype implementation of this design.

1 Introduction

In this paper, we apply tools including peer-to-peer computing and secure coprocessors to distribute the SEM approach to PKI, and thus preserve its advantages while overcoming its scalability, reliability, and trust problems. Sect. 2 reviews the SEM approach, and discusses its advantages and disadvantages. Sect. 3 discusses the tools we apply to this problem. Sect. 4 discusses the design we build with these tools. Sect. 5 discusses our prototype. Sect. 6 discusses some related approaches. Sect. 7 discusses some conclusions and future work.

2 SEM

Motivation In PKI, a *certificate* is a signed assertion binding a public key to certain properties. The correctness of the trust decisions a relying party makes depends on the assumption that the entity knowing the matching private key possesses those properties. When this binding ceases to hold, this certificate needs to be *revoked*, and this revocation information needs to propagate to relying parties, lest they make incorrect trust judgments regarding that public key.

* This work was supported in part by the Mellon Foundation, by the NSF (CCR-0209144), by Internet2/AT&T, and by the Office for Domestic Preparedness, U.S. Dept of Homeland Security (2000-DT-CX-K001). The views and conclusions do not necessarily represent those of the sponsors.

Consequently, fast and scalable certificate revocation has been area of active research in recent years (e.g.,[21, 23]). In their *Security Mediator*[1] *(SEM)* approach, Boneh et al [4] proposed a system that revokes the ability of the keyholder to use a private key, instead of (or in addition to) revoking the certificate attesting to the corresponding public key.

Architecture The SEM approach is based on *mediated RSA (mRSA)*, a variant of RSA which splits the private key of a user into two parts. As in standard RSA, each user has a public key (n_u, e_u) and a private key d_u, where n is the product of two large primes, $\gcd(e_u, \phi(n_u)) = 1$, and $d_u * e_u = 1 \pmod{\phi(n_u)}$. The public key of a user u is the same as in standard RSA, as is the public-key operation. The two parts of a user's secret key are $d_{\text{sem},u}$ and $d_{\text{user},u}$, where d_u is the standard secret key and $d_u = d_{\text{sem},u} + d_{\text{user},u} \pmod{\phi(n_u)}$. $d_{\text{user},u}$ is the part held by the user and $d_{\text{sem},u}$ is the part held by the SEM. (We note that $d_{\text{sem},u}$ and $d_{\text{user},u}$ are each statistically unique for each user u.)

This division of the secret key requires changes to the standard RSA key setup because a SEM must not know $d_{\text{user},u}$ and a user must not know $d_{\text{sem},u}$. So, a trusted party (e.g., a CA) performs key setup by generating a statistically unique $\{p_u, q_u, e_u, d_u, d_{\text{sem},u}\}$ for a user u. The private key d_u is generated in the standard manner, but is communicated to neither the SEM nor the user. Instead, $d_{\text{sem},u}$ is chosen as a random integer in $[0, n_u - 1]$, and $d_{\text{user},u}$ is then calculated as $d_{\text{user},u} = d_u - d_{\text{sem},u} \pmod{\phi(n_u)}$.

Because the private key d_u is split into two "halves," private key operations require the participation of both the user and the SEM: e.g., each party raises the message to its half-exponent, modulo n, and the results are then multiplied, also modulo n. (See Fig. 1.) Thus the full private key never needs to be reconstructed

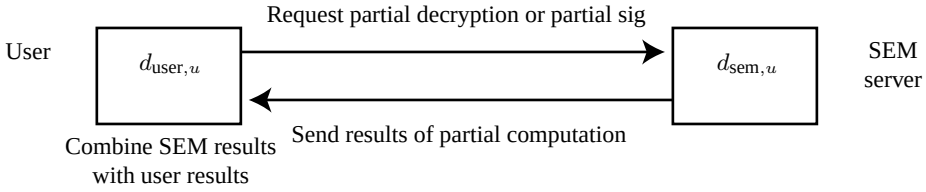

Fig. 1. The general SEM algorithm.

Advantages The SEM approach provides several advantages. Since these essentially are standard RSA operations, a SEM PKI is compatible with most legacy public-key cryptography tools. Since a full private-key operation can occur only if the SEM believes the user's key pair is valid, then the system can revoke a key pair by having the SEM refuse to carry out its half of the operation. This approach can reduce or even eliminate (in the case of revocation due to administrative action, such as a user ceasing employment) the need for certificate revocation lists—since a private-key operation (such as signature or decryption) cannot occur after revocation.

[1] Also referred to as "semi-trusted mediator."

Furthermore, the SEM itself gains no useful information in servicing users. When decrypting, the SEM receives the ciphertext but is only able to partially decrypt it, so no useful information could be gained by a malicious SEM. For signature generation, a user sends the SEM a hash of the message which the SEM uses to generate the signature. This also contains no information about the cleartext of the message itself, so a user's data is kept confidential.

Additionally, the compromise of a single SEM does not compromise the secret keys of any users. Instead, the attacker is able to revoke the security capabilities for users connected to the SEM. Although Boneh et al state that attackers could unrevoke revoked certificates, this can be prevented by having honest SEMs permanently delete $d_{\text{sem},u}$ upon revocation.

Disadvantages However, the initial SEM approach has scalability disadvantages. In a large-scale distributed system, we must allow for problems such as mobile users, network partitioning, crashing of machines, and occasional compromise of servers. To accommodate a large population, we could add multiple SEMs. However, if a user's $d_{\text{sem},u}$ lives on exactly one SEM, then we have many problems: temporary denial of service if the network is partitioned; permanent denial of service if the SEM suffers a serious failure; inability to revoke the key pair if an adversary compromises a SEM and learns its secrets.

In their original paper, Boneh et al did propose one way to distribute the SEM architecture by using a stateless model in which a user can connect to any SEM. However, this approach required that the entire SEM network have a single RSA key pair, so that any node can access the encrypted $d_{\text{sem},u}$ bundled with each request. This network-wide key pair could either be stored on each island through replication or shared securely among islands using threshold cryptography. In the first case, compromise of a single island is potentially easier (due to replication) and causes damage to the entire network. In the latter case, each user request requires distributed computation among islands which hurts performance. In either case, the user is at risk if she connects to a compromised SEM.

3 Tools

To address the problem of distributing SEM, we use several tools.

We need to be able to trust a SEM to use and delete each user's $d_{\text{sem},u}$ when appropriate, and not transmit it further. However, the more we distribute the SEMs throughout a network, the less foundation we have for such trust. To address this problem, we use *secure coprocessors*, such as the IBM 4758 [26]. This gives us a general-purpose computing environment and cryptographic protections, coupled with high-assurance protection against physical attacks, and an *outbound authentication* scheme which lets software applications running on the coprocessor authenticate themselves to remote parties [25]. This platform thus gives us a safe and confidential environment in remote environments. If a user trusts our software is not flawed, then the user can also trust that software executed cannot be altered by adversaries and the user may also remotely authenticate instances of this software. (In Sect. 7, we consider using newer trusted computing platforms as well.)

We'd like to make it easy for users to find SEMs (and for SEMs to find each other), and we'd like this functionality to persist despite failures and (potentially) malicious attacks. *Peer-to-peer networking (P2P)* (embodied by technology such as Gnutella) is an attractive choice here. With P2P, communication does not rely on a central entity to forward requests or messages. Rather, each entity either tries to satisfy a request itself, or forwards it to its neighbors, in the spirit of the older distributed concept of *diffusing computation* [2].

This decentralization is a key benefit of the peer-to-peer network, as it removes any central entity necessary for the system to function. Without a central controlling server, the network's survivability increases by causing denial of service attacks to be much more difficult (as the RIAA has found to its dismay). Additionally, the damaging effects of network partitions are potentially alleviated by standard P2P communication algorithms, as a new path to a destination may be found.

We will also need to distribute critical secrets across multiple SEMs, for resilience against attack. Here, we can use the standard technique of *threshold cryptography* [24]. Given a secret y and parameters $t < k$, we construct a degree t polynomial that goes through the point $(0, y)$, and choose k points on this polynomial as *shares* of y. Any t shares suffices to reconstruct the polynomial and hence y, but fewer than t shares give no information.

We need to accommodate the fact that machines may be compromised, and the secrets they store may become exposed to the adversary. To mitigate the damage of such potential exposure, we can use the technique of *strong forward security* [5]. We divide time into a sequence of clock periods, and use a cryptographic system such that even if the private key for a given period is exposed, use of the private key in previous or future sessions is still secure. Burmester et al give two examples of strong forward secure schemes, one for any public key cryptosystem and another for use in an El Gamal key escrow system.

4 Design

Architecture In our basic architecture, we envision SEMs as trustworthy *islands* distributed throughout the network. We use a secure coprocessor to house each SEM and thus give it a foundation for this trustworthiness. As noted earlier, this technology also lets each island have a key pair and certificate chain that establishes the entity who knows the private key is an instantiation of our island software on an untampered device. Thus, users can authenticate islands, and islands can authenticate each other.

Each island will house resources that enable it carry out services. When a user requests such a service, we use P2P techniques to carry the request to the proper island, and carry the response back to the user. (As we discuss below, individual islands will also house resources other islands need; we can use P2P there as well.)

Despite physical protections, an individual island may still become compromised and reveal its data to the adversary. An individual island may also become unavailable, due to crash or partition. To handle these scenarios, we build a *migration* scheme based on threshold cryptography and strong forward security.

When we initially create a secret x and transmit it to an island L, we also split into k shares using threshold cryptography. We securely transmit each share of x to a different island. (Additionally, the shares may be proactively updated using the techniques described in [11, 12, 14, 15] so that an attacker may not slowly acquire enough shares to reconstruct x.) After those steps are complete, the secret is stored both on the primary island L and on k other islands, so an attacker must either compromise L or compromise t of the k islands in order to get x.

When island L is unavailable to fulfill a request that requires x, then the requester will have to be redirected to another island M, and the shareholders will need to participate in reconstructing x there. However, since the original island L may have been compromised, x must be updated using strong forward security so that the old version on L is rendered useless.

The general migration scheme is executed as follows:

1. The user tries to connect to the assigned island L, but fails.
2. The user then connects to another island M instead. M may be chosen either pseudorandomly, using a load balancing algorithm or another scheme (e.g., based on network proximity to the user).
3. The islands that hold shares of x are contacted and, and this x is updated using strong forward security. As discussed below, this update may or may not involve reconstruction of x, depending on the method chosen. Generally, the strong forward security scheme will vary depending on the how the secret x is generated.
4. Strong forward security results in M storing the updated secret.
5. Migration is complete and M can then fulfill the user's request.

SEM Operations To use this architecture for SEM, each island acts as a SEM mediator, holding $d_{\text{sem},u}$ for a number of clients. We distribute load across the islands by, at key generation, assigning users to different SEMs. (We could also distribute load via migration.) As with the original SEM architecture, a user's $d_{\text{sem},u}$ is stored in full only on one island.

In the original SEM scheme [4], a CA generates key pairs for users and splits d into two halves. In our variant, the CA must additionally share $d_{\text{sem},u}$ to k islands in the network using threshold cryptography. (See Fig. 2 Also stored with those shares is the user's identity and the revocation status of the user's key pair (initialized to "false," not revoked). For key generation, the CA must be able to prove its identity to the islands; otherwise, the islands will ignore its request. If we desired an escrow service to allow authorized decryption of data after revocation (or if we do not decide to use the CA during migration, as discussed below), we also distribute shares of the full secret key d_u.

If the island that holds $d_{\text{sem},u}$ and revocation information for a user u goes down, then the other islands must be able to determine whether the user's key pair has been revoked. We accomplish this by, during revocation, having the shareholders as well as original island update the revocation status for that key pair. In our initial vision, we delete the shares of $d_{\text{sem},u}$ that are stored on k other islands.

Assume that a user's $d_{\text{sem},u}$ is stored on island L. The network is notified that a user's key pair is to be revoked, and a P2P request is generated to L to revoke the user's

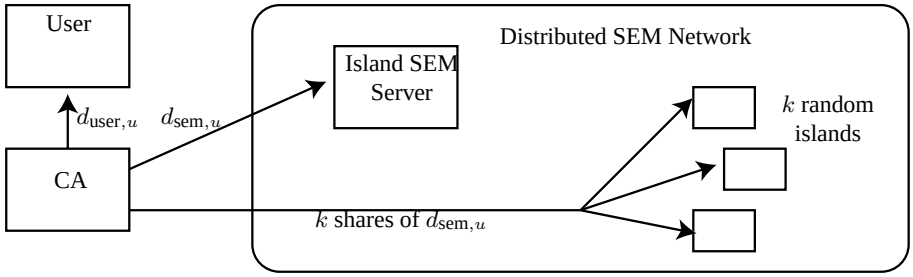

Fig. 2. Key set-up in our proposed system

key pair. If L is operational, then L notifies all of the other islands that hold shares of $d_{\text{sem},u}$ to delete them and store the fact that $d_{\text{sem},u}$ has been revoked; L also deletes $d_{\text{sem},u}$ and adds the user's serial number to its CRL. Else if L is not operational, then the k islands holding the shares of $d_{\text{sem},u}$ are notified and told to delete their shares. (Note that in this case, migration—see below—has not yet occurred for this user or else an island would have been contacted.)

SEM Migration If a user u issues a request but the island L holding $d_{\text{sem},u}$ is not available, then we select another island M and request migration. As with other selections, M may be chosen in a number of ways (although a random or pseudorandom way, so that an attacker cannot predict it, would help in some scenarios—we plan to add this in future work). See Fig. 3.

After that initial step is performed, we have two different approaches, depending on whether a CA exists that can know the full private key d_u. Any communication between the islands is authenticated using the outbound authentication of the secure coprocessors and it is assumed that the online CA also has some mode of outbound authentication to prove the source of its messages.

For added resilience, we can have shareholder islands not participate in migration if they can still ping the original island L.

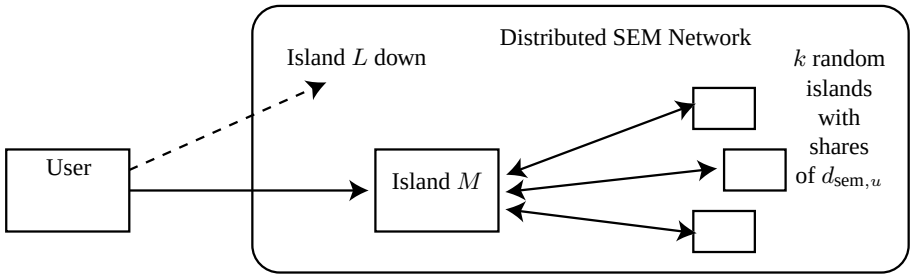

Fig. 3. Migration in our proposed system

If we have an online CA, the new island M contacts it and tells it that migration is to occur. (If we constrain the choice of the next M, then the CA must verify that M is a satisfactory candidate.) M sends a request for any islands in the network that have shares of $d_{\text{sem},u}$ to send those securely to the CA using standard RSA encryption. The CA can then reconstruct $d_{\text{sem},u}$. If the CA stores the full private key for d_u, then it can use that. Otherwise, the shareholders for d_u must also send those, so the CA can reconstruct it as well. The CA generates a random number x in the interval $[0, n_u - 1]$ such that $x \neq d_{\text{sem},u}$, and calculates $y = d_u - x \pmod{\phi(n_u)}$. The CA securely distributes shares of x to the k shareholder islands using threshold cryptography; the shareholder islands delete the old shares for $d_{\text{sem},u}$. The CA securely sends y to the user by encrypting it with the user's public key and then partially decrypting with the old $d_{\text{sem},u}$. The user deletes the old $d_{\text{user},u}$ and sets $d_{\text{user},u} = y$. The CA securely sends the x to M, who deletes the old $d_{\text{sem},u}$ and sets $d_{\text{sem},u} = x$.

Once the user receives y, it reconnects to the network and performs the operation again, using its new value $d_{\text{user},u}$. At this point, M can complete the request and the migration is complete.

If no CA exists, then we need to generate a new $d_{\text{sem},u}, d_{\text{user},u}$ pair without reconstructing d_u or $\phi(n_u)$, since we do not have a safe place to store them.

For now, we borrow the trick of [16] and have M generate a δ in a range $[-r, r]$, and changing $d_{\text{sem},u}$ to $d_{\text{sem},u} - \delta$, where r is big enough to keep the key halves changing unpredictably, but small enough to be smaller than $d_{\text{sem},u}$ and $d_{\text{user},u}$ for a practically indefinite number of rounds. M sends δ to the user (encrypted with u's public key and then partially decrypted with the old $d_{\text{sem},u}$); the user replaces $d_{\text{user},u}$ with $d_{\text{user},u} + \delta$. (We could also use the [16] trick of having M and u together pick r, to reduce risk from a compromised M.)

This way, neither M nor u need to know $\phi(n_u)$, but the new $d_{\text{user},u}$ and $d_{\text{sem},u}$ remain positive and still sum to d_u. M splits r into k shares and sends each to a $d_{\text{sem},u}$ shareholder; each shareholder uses its piece to to update its share.

It is tempting to have M pick a new $d_{\text{sem},u}$ directly and distribute shares to the shareholders of d_u, who then calculate the new $d_{\text{user},u}$ in a nicely distributed fashion. However, as of this writing, we can cannot see how to reduce $d_u - d_{\text{sem},u}$ to $d_u - d_{\text{sem},u} \pmod{\phi(n_u)}$ without reconstructing $\phi(n_u)$.

Once the user receives the new $d_{\text{user},u}$, it can compute its half of the normal computation using the new $d_{\text{user},u}$. At this point, M can also complete its half of the computation because it has generated a new $d_{\text{sem},u}$ and the migration is complete.

As an area of future work, we are also considering incorporating strong forward secrecy into regeneration of the user's private key during regeneration of $d_{\text{sem},u}$. We already have trusted hardware, one of the components of some SFS schemes in the literature (e.g., [32, 9]). Furthermore, this would protect against compromise of L by the user u, in order to obtain $d_{\text{sem},u}$ and reconstructing d_u.

When an island goes down (or is compromised and subsequently shut down and restarted from a clean state), it has a few options upon reboot:

The island could delete all of the key halves it has stored, and thereby force users to migrate back to it. New users would also be assigned to it. It also deletes all of the shares of that it stored and requests new shares of those to be generated.

Alternatively, the island could poll the other islands to determine which $d_{\text{sem},u}$ halves have migrated away from it. It then deletes the information for the users that migrated away and continues serving the other users. The island can continue using the shares it has stored, but it must determine whether any of them are out of date. Since the $d_{\text{sem},u}$ shares must be updated during migration, the $d_{\text{sem},u}$ shares could be invalid and the network must be polled to determine whether this is the case. If so, then the outdated shares must be updated.

Analysis First, we consider compromise of specific entities.

If the CA has been compromised, then we have a serious problem, since the CA generates the users' initial key pairs, and in the CA-migration case, learns the new key pairs as well.

Alternatively, suppose L has been compromised. The island L holds $d_{\text{sem},u}$ for some number of users. Additionally, L stores shares for some other users in the distributed SEM network. We must assume that the attacker has access to all of these values, so now we analyze what privileges are granted by illicit access to them.

- If the attacker acquires $d_{\text{sem},u}$ for another user, then migration effectively disables this $d_{\text{sem},u}$ because it causes a new $d_{\text{user},u}$ to be issued to the user. (Also, note that the new $d_{\text{user},u}$ is not sent back to L.) Since the new $d_{\text{user},u}$ does not mesh with the old $d_{\text{sem},u}$ due to the mRSA protocols, the old $d_{\text{sem},u}$ is rendered useless.
- If the attacker acquires $d_{\text{sem},u}$ and colludes with that user, then the attacker will be able to compute d_u from $d_{\text{user},u}$ and $d_{\text{sem},u}$, so migration fails to achieve full security in this case (unless, as discussed earlier, we try implementing SFS here as well).
- If the attacker colludes with a user whose key-half is not on that island, then the user and attacker might trick the SEM network to migrating that user's data to L, and thus reconstruct $d_{\text{sem},u}$. The user will then be able to reconstruct d_u, the full private key. This problem can be mitigated by using pings (as stated in section 4.3) to ensure that the user's main island is unavailable, and also using a non-predictable way to generate the next island for migration (to reduce the chance that M is a valid candidate). However, such problems may be the inevitable cost of higher availability in the distributed SEM network.
- If the attacker acquires shares of a $d_{\text{sem},u}$, the attacker effectively acquires no valuable information, unless the attacker also gains access to enough other shares of either in order to reconstruct them. However, this can be made extremely difficult using proactive share updating, as described in [11, 12, 14, 15].

Clearly, there is a period between the time of compromise time when that compromise is discovered. However, the use of secure coprocessors causes any physical attacks to be detected immediately (with high assurance) and stopped by the zeroization of coprocessor data.

If M is compromised, then the attacker gets access to the new $d_{\text{sem},u}$, so the migration is unsuccessful. However, as long as another migration to an uncompromised island can be performed, the $d_{\text{sem},u}$ acquired by the attacker can be rendered useless as described above. Additionally, in this case the attacker could send the user fake data for

the new $d_{\text{user},u}$, but any resulting inconsistencies with decryption or signature generation would just flag M as compromised. As stated above, with secure coprocessors the window of time before this compromise is discovered should be small.

Network Trust Model The primary parties that require use of the network are the islands that comprise the network itself. Each island must have exclusive access to certain services in order to provide fast revocation of security capabilities. However, the CA (and users and islands) must also be able to gain access to the network services in a restricted way. Clearly, the requests of each party will differ and there must be a clear delineation of capabilities between them. For example, during migration, islands will have to search the network to find other islands that hold the shares of a user's $d_{\text{sem},u}$. Although this operation can be executed by the CA as well (when CAs can perform migration), users should not be able to (easily) determine the location of or acquire shares.

Islands join the network normally and become full members of it. Since each island in the network has a secure coprocessor with outbound authentication/attestation, each member in the peer-to-peer network can prove that it is a trusted island with certain privileges. This creates a trust network in which each island is known to be executing unmolested as long as our software is not flawed (and the coprocessor's physical security protection works). Once an island has authenticated itself to the system, it can search the network, advertise services, and perform any other command allowed by the peer-to-peer software.

Certificate Authorities can interact with the network in one of two ways: (1) they can connect to an island server that provides an interface to the rest of the network; or (2) they can connect directly to the P2P network, but with limited capabilities (registration and, if implemented using a CA, then migration). For example, the CA must be able to somehow query the network during key generation to determine to which island to assign the user.

Users do not connect directly to the P2P network, but instead communicate with an island that provides indirect access to the services available on the network. For example, during normal operation, users connect directly to their assigned island, but if that fails, then the user must notify the P2P network that migration is necessary. Users do so by connecting to another island (available in a public list) and requesting the migration service. The user is then assigned to an island and further communication occurs directly between that island and the user.

5 Prototype

We are currently building a prototype of our Distributed SEM approach; at the time of this writing, our current code (2000 lines of Java) deals with the peer-to-peer aspects of key generation and execution of migration (See Fig. 4.)

The *Island Server Code* performs migration and the island's part of decryption and key and signature generation using $d_{\text{sem},u}$. This part also introduces the networking support for the islands. It is a combination of both the original SEM server code, along with our server-related migration code.

The code on each island is divided into two parts. The *P2P network code* consists of the peer-to-peer access layer and the protocols necessary for island communication.

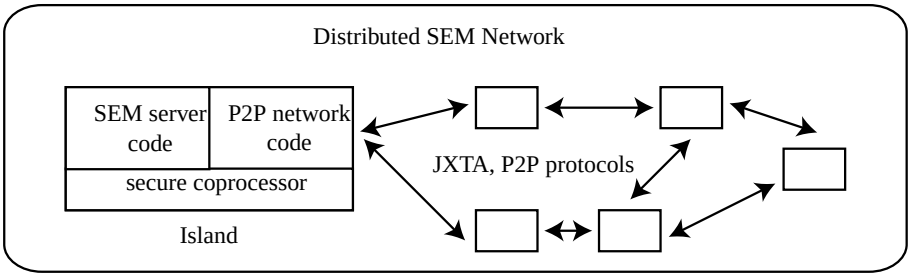

Fig. 4. Architecture of our system

We use the Project JXTA open source framework [29] to accomplish this. Each island runs a server that accepts connections from only the SEM server on the same machine. The server application is built on top of JXTA and uses the peer-to-peer protocols, such as searching and pinging, available. It executes the distributed aspects of key generation (distribution of shares of $d_{\text{sem},u}$) and migration.

Specifically, during the distributed SEM operations we use JXTA's discovery service to find islands with the information needed (e.g. shares of $d_{\text{sem},u}$). We plan to leverage the security features of the JXTA framework to secure the P2P activities of the distributed SEM network. These include secure P2P groups to restrict access to the network and secure pipes to allow safe distribution of shares to an island during migration. Furthermore, we envision the use of outbound authentication data of secure coprocessors [25] in JXTA XML messages to validate the source of messages generated in the migration and revocation algorithms. The capabilities of the secure coprocessors will be exposed in the Java code using the Java Native Interface (JNI) [27].

The *SEM server code* accepts requests from users and handles most of those without using the P2P layer. When a migration or key registration request is received, however, the server code forwards the request to the internal server running JXTA and the internal server completes the request.

This is a modular approach that allows us to change the peer-to-peer implementation in the future. In the current prototype, we have implemented our network code in a simplified version. (Integration with the SEM server code, utilization of security features in the P2P layer, and porting on to the 4758 remain to be done.)

The *Certificate Authority Code* participates in key generation, as in the original SEM architecture. With our additions it is necessary for the CA to connect to an island and initiate a P2P registration request. We have implemented the code to perform registration in the network. (However, it is not yet integrated with the original SEM key generation code.)

The *Client User Code* combines the functionality of the original SEM architecture, which consisted of the user's half of signature generation and decryption, along with with the additional steps required to request migration and process the migration response. (This remains a task for the future.)

Our code will be available for public download. Since the original SEM code is covered under the GPL, our changes—for migration and support of distributed functionality—are as well.

6 Related Work

Trusted Hardware and P2P Marchesini and Smith [18] built a *hardened Gnutella* P2P communication system within secure coprocessors; Boneh et al [13] also considered the potential of combining trusted computing with P2P.

Strong Forward Secrecy Tzeng and Tzeng [32] considered strong forward security with threshold cryptography for El Gamal signatures.

Standard Revocation Techniques *Certificate Revocation Lists* and variations on this method (e.g., Δ-CRLs) are one of the most common methods of certificate revocation. To revoke a certificate, the CA adds the serial number of the revoked certificate to the CRL and then publishes that CRL to a directory. Since this is only done periodically, CRLs are not a guarantee that a certificate is still valid. Also, since CRLs may be very large, users will generally not want to have to download them very often. In order to check whether a certificate is revoked, a user must potentially download a long CRL. To mitigate this problem, Δ-CRLs only distribute a list of the certificates revoked since the last CRL was distributed. Additionally, Cooper notes that when a new CRL is issued there will be a peak time in which many requests are made to download the CRL from the directory (because everyone wants to make sure that certificates aren't revoked and a CRL expires at the same time for everyone). He suggests spreading out the requests for CRLs over time by "over-issuing" CRLs such that a new CRL is published before the old one expires [7].

Another similar technique is *Windowed Key Revocation*, which uses CRLs but with a twist that certificates are assumed to be valid for a certain "window" of time and that CRLs have a reduced size due to the revocation window [20]. Additionally, in this scheme verifiers can control the allowed "window" time and to check if a certificate is revoked, the verifier checks the windowed CRL issued or grabs a new certificate from the CA.

The *Online Certificate Status Protocol (OCSP)* provides online verification that a certificate is still valid. This requires a CA to generate a digital signature for each request because the response from the CA must be signed [21]. The CA stores an internal log of the status of all certificates or possibly just a CRL that it doesn't publish. So, addition of revoked certificates is quick and the certificate status is updated instantly. A user must be online and must connect to the CA and check the status of a certificate.

Certificate status verification is a computationally expensive operation, as the response from the CA must be digitally signed. If a single validation server performs OCSP, then all requests must be routed to it, potentially overloading the server. Security may be weakened by a distributed environment because if any keys of any OCSP servers are compromised, then the entire system is compromised.

With *Certificate Revocation Trees*, instead of keeping entire list of revoked serial numbers, we keep a list of ranges of serial numbers that are good or bad. This saves

space (better than standard CRLs), but adding a serial can involve a good amount of computation as it can require the entire tree to be recomputed. Still, revocation status can be quickly determined by a fast search through the tree.

Newer Revocation Techniques In [4], Boneh et al give a discussion of the benefits of the original SEM architecture with regards to other current solutions. Since the publication of that paper, a few other techniques for certificate revocation have been developed. Micali's *NOVOMODO* approach [23] uses one-way hashing and hash chains to show the validity or revocation status of certificates. Centralized NOVOMODO—in which the central secure server responds to all validity requests—is prone to performance issues and denial of service attacks as it is the central source for certificate validity proofs. Micali also presented a distributed version: having one central trusted server send out an array of the current validity proofs for all users to each server in the network. Micali does not discuss solutions to many potential problems that could occur during a distribution, such as bandwidth problems, network partition or untrusted server corruption. Micali states that it should be very difficult to attack the central server, as it does not accept incoming requests, but an attacker could instead attack the network surrounding the server, preventing it from distributing the array. In other words, distributed NOVOMODO still has a central "head" that can be severed (albeit in a more difficult way) in order to shutdown the system.

Ding et al introduce *Server-Aided Signatures (SAS)*, [8] a technique based on mediated cryptography similar to the SEM architecture with the focus on minimizing client computation load. While Distributed SEM works with both signature generation and decryption, SAS only deals with signature generation. It achieves a performance boost for the user by only requiring the user to compute a hash chain during setup, in a similar fashion to NOVOMODO, but differing in that the user keeps the chain secret. In SAS, the server must be stateful and, for each user, must save their certificate, i, and all of the signatures already generated for that user. This amount of state makes migration infeasible as every signature would have to be distributed on other islands using threshold cryptography. Additionally, in SAS the corruption of a server allows the attacker to produce user signatures because all of the prior signatures are saved on the server.

SEM Tsudik [31] and Boneh et al [3] have also followed up on their original SEM work. [16] explores adding proactive updates to the key halves.

7 Conclusions and Future Work

In this paper we have introduced a method to distribute SEM by using a network that combines the benefits of secure coprocessors and peer-to-peer networking, and provides providing efficient and uninterrupted access to private data stored on a trusted third party, even in the event of occasional server compromise. This approach avoids replication of data across the network while also avoiding the common use of distributed computation in order to access the secrets stored.

Once the implementation is completed, an area of great interest will be performance testing and tuning. The performance of both migration itself and the entire application

running on the full P2P network (using secure coprocessors) will be reveal much information about our approach to distributed SEM—and the feasibility of this P2P/trusted hardware network.

Also, the IBM 4758 is a relatively expensive special-purpose device. Recent advances in *trusted computing*—both with COTS hardware (e.g., [10, 19, 30]) as well with experimental CPUs (e.g., [6, 17, 22, 28])—explore the potential of achieving similar functionality (albeit a lower level of physical security) in standard desktop platforms. We plan to explore the potential (and relative performance) of distributed SEM on these platforms as well. We also plan to use our framework of P2P on trusted hardware to explore other applications as well. Finally, general Byzantine attacks must be considered in the Distributed SEM network and extra steps (e.g., [1]) must be taken to ensure the correct completion of all operations.

References

1. Alon, N., Kaplan, H., Krivelevich, M., Malkhi, D., Stern, J.: Scalable Secure Storage When Half the System Is Faulty. Information and Computation **174** (2002) 203–213
2. Andrews, G.: Paradigms for Process Interaction in Distributed Programs. ACM Computing Surveys **23** (1991) 49–90
3. Boneh, D., Ding, X., Tsudik, G.: Fine-Grained Control of Security Capabilities. ACM Transactions on Internet Technology (2004)
4. Boneh, D., Ding, X., Tsudik, G., Wong, C.M.: A method for fast revocation of public key certificates and security capabilities. In: 10th USENIX Security Symposium. (2001) 297–308
5. Burmester, M., Chrissikopoulos, V., Kotzanikolaou, P., Magkos, E.: Strong Forward Security. In: IFIP-SEC '01 Conference, Kluwer (2001) 109–121
6. Chen, B., Morris, R.: Certifying Program Execution with Secure Processors. In: 9th Hot Topics in Operating Systems (HOTOS-IX). (2003)
7. Cooper, D.A.: A model of certificate revocation. In: Fifteenth Annual Computer Security Applications Conference. (1999) 256–264
8. Ding, X., Mazzocchi, D., Tsudik, G.: Experimenting with server-aided signatures. In: Network and Distributed Systems Security Symposium. (2002)
9. Dodis, Y., Katz, J., Xu, S., Yung, M.: Strong Key-Insulated Public-Key Schemes. In: Public Key Cryptography—PKC 2003, Springer-Verlag LNCS 2567 (2003) 109–121
10. England, P., Lampson, B., Manferdelli, J., Peinado, M., Willman, B.: A Trusted Open Platform. IEEE Computer (2003) 55–62
11. Frankel, Y., Gemmell, P., MacKenzie, P.D., Yung, M.: Optimal resilience proactive public-key cryptosystems. In: IEEE Symposium on Foundations of Computer Science. (1997) 384–393
12. Frankel, Y., Gemmell, P., MacKenzie, P.D., Yung, M.: Proactive RSA. In: Advances in Cryptology—CRYPTO 97, Springer Verlag LNCS 1294 (1997) 440–454
13. Garfinkel, T., Rosenblum, M., Boneh, D.: Flexible OS Support and Applications for Trusted Computing. In: 9th Hot Topics in Operating Systems (HOTOS-IX). (2003)
14. Herzberg, A., Jakobsson, M., Jarecki, S., Krawczyk, H., Yung, M.: Proactive public key and signature systems. In: ACM Conference on Computer and Communications Security. (1997) 100–110
15. Herzberg, A., Jarecki, S., Krawczyk, H., Yung, M.: Proactive secret sharing or: How to cope with perpetual leakage. In: Advanced in Cryptology—CRYPTO 95, Springer Verlag LNCS 963 (1995) 339–352

16. Le, Z., Smith, S.: Proactive mediated rsa. Manuscript, Department of Computer Science, Dartmouth College (2004)
17. Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., Horowitz, M.: Architectural Support for Copy and Tamper Resistant Software. In: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems. (2000) 168–177
18. Marchesini, J., Smith, S.: Virtual Hierarchies: An Architecture for Building and Maintaining Efficient and Resilient Trust Chains. In: Proceedings of the 7th Nordic Workshop on Secure IT Systems—NORDSEC 2002, Karlstad University Studies (2002)
19. Marchesini, J., Smith, S., Wild, O., Macdonald, R.: Experimenting with TCPA/TCG Hardware, Or: How I Learned to Stop Worrying and Love The Bear. Computer Science Technical Report TR2003-476, Dartmouth College (2003)
20. McDaniel, P., Jamin, S.: Windowed certificate revocation. In: 2000 IEEE Symposium on Security and Privacy. (2000) 1406–1414
21. McDaniel, P., Rubin, A.: A response to can we eliminate certificate revocation lists? In: Financial Cryptography. (2000)
22. McGregor, P., Lee, R.: Virtual Secure Co-Processing on General-purpose Processors. Technical Report CE-L2002-003, Princeton University (2002)
23. Micali, S.: Novomodo: Scalable certificate validation and simplified pki management. In: 1st Annual PKI Research Workshop. (2002)
24. Shamir, A.: How to share a secret. In: Communications of the ACM. Volume 22. (1979) 612–613
25. Smith, S.: Outbound Authentication for Programmable Secure Coprocessors. In: Computer Security—ESORICS 2002, Springer-Verlag LNCS 2502 (2002) 72–89
26. Smith, S., Weingart, S.: Building a High-Performance, Programmable Secure Coprocessor. Computer Networks **31** (1999) 831–860
27. Stearns, B.: Trail: Java Native Interface. Sun Microsystems, Inc. (2004) `http://java.sun.com/docs/books/tutorial/native1.1/`.
28. Suh, G., Clarke, D., Gassend, B., van Dijk, M., Devadas, S.: AEGIS: Architecture for Tamper-Evident and Tamper-Resistant processing. In: Proceedings of the 17 International Conference on Supercomputing. (2003) 160–171
29. Sun Microsystems, Inc.: Project JXTA: Java Programmers Guide. (2001) http://www.jxta.org.
30. Trusted Computing Platform Alliance: TCPA PC Specific Implementation Specification, Version 1.00. `http://www.trustedcomputinggroup.org` (2001)
31. Tsudik, G.: Weak Forward Security in Mediated RSA. In: Security in Computer Networks Conference. (2002)
32. Tzeng, Z., Tzeng, W.: Robust Key-Evolving Public Key Encryption Schemes. Crypology Eprint Archive Report `http://eprint.iacr.org/2001/009` (2001)